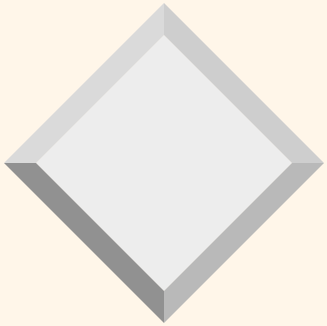


Managing Storage: Above the Hardware

Where we are

- Last time: hardware
  - HDDs and SSDs
- Today: how the DBMS uses the hardware to provide fast access to data

How DBMS manages storage

- "Bottom" two layers are **disk space manager** and **buffer manager**

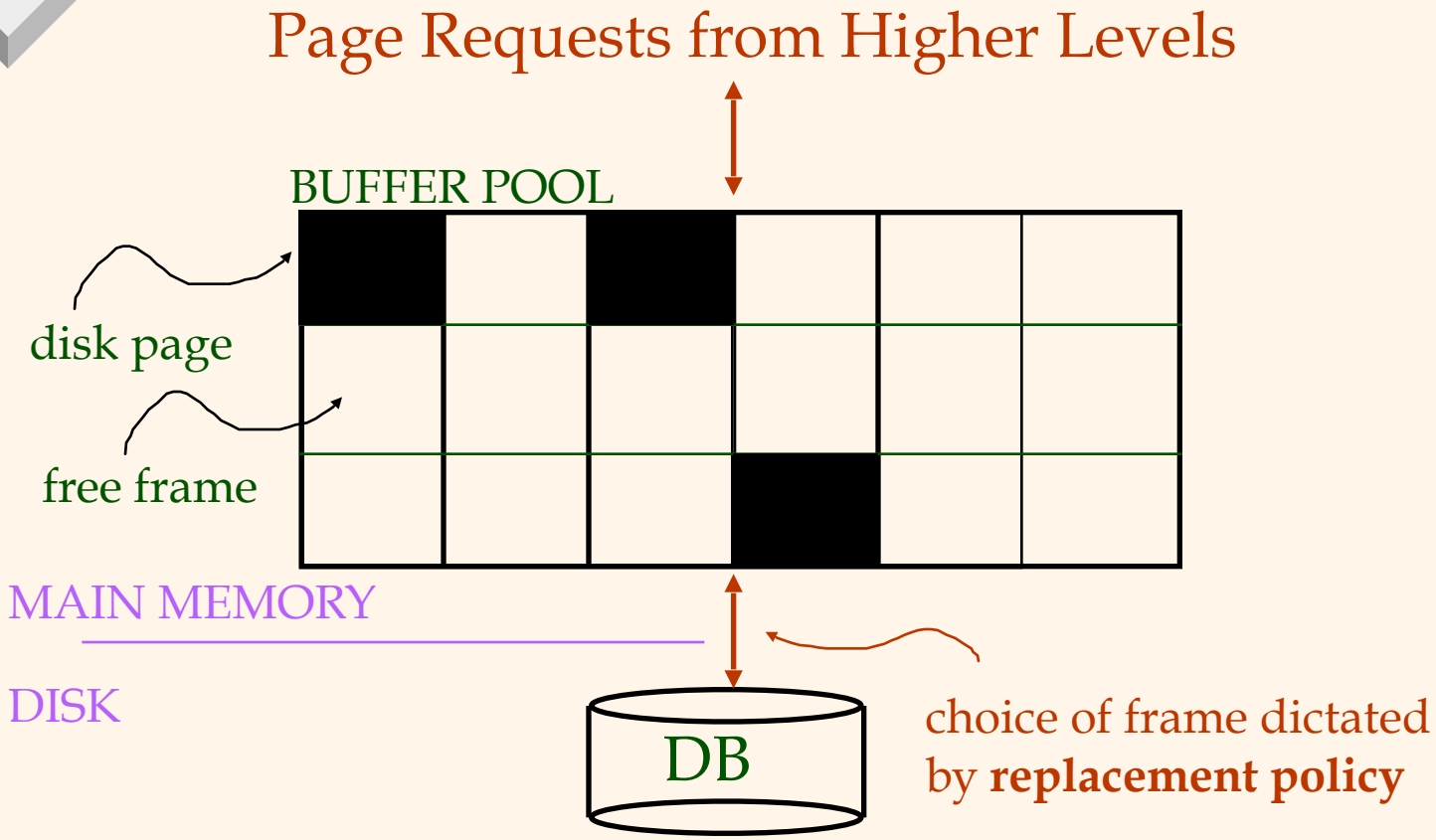
Disk Space Manager

- Supports the concept of a *page* – a unit of data the size of a disk block
- Provides commands to allocate/deallocate/read/write a page
- Tries to ensure pages that will be accessed together are stored contiguously if possible
- May use (or extend) OS filesystem facilities

Buffer Manager

- This is the component responsible for controlling when pages move between memory and disk
- DBMS has a buffer pool – an area of RAM which will be used to cache data as it goes to/from disk
  - You can set the size (see documentation)
  - Buffer pool divided into **frames** (page-sized slots)
- Buffer Manager will bring data in/out of memory "smartly" to maximize performance

Buffer Management in a DBMS



- *Data must be in RAM for DBMS to operate on it!*
- *Table of <frame#, pageid> pairs is maintained.*

Wait a second…

- Doesn't the OS already do something similar with virtual memory?
- Yes, but DBMS usually doesn't want to rely on OS *exclusively*
  - DBMS knows its own unique access patterns and can better decide what data to fetch/prefetch
  - DBMS needs ability to force some data to disk to guarantee fault-tolerance (later in the course)
- Typically, will still interact/interface with OS virtual memory functionality

When a Page is Requested ...

- If page is not in pool (cache miss):
  - Choose a frame for *replacement*
  - If frame contains a page with changes, write it to disk
  - Read requested page into chosen frame
  - *Pin* the page and return its address.
- If requested page is in pool (cache hit):
  - Increment its pin count and return its address.

- *If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time*

More on Buffer Management

- Page in pool may be requested many times,
  - a *pin count* is used. A page is a candidate for replacement iff *pin count* = 0.
- Requester of page must *unpin* it when it is done with it, and indicate whether page has been modified:
  - *dirty* bit is used for this.

Questions to think about

- What happens if the buffer is full and all frames have pin count > 0?

- What happens if multiple transactions (users) want to access the same page?

Buffer Replacement Policy

- Frame is chosen for replacement by a *replacement policy*
- This policy allows us to determine which of the available frames to use (if there is more than one)
  - I.e. which page to evict back to disk
- Goal: minimize number of cache misses
  - If we evict a page that's needed again in the future, cache miss next time someone asks for it.

Buffer Replacement Policies

- Optimal algorithm to minimize number of cache misses: **farthest-in-future** (evict page which won't be needed for longest time in future)
  - Proof somewhat subtle, beyond scope of this class
  - Problem: requires knowledge of the future
- Least-recently-used (LRU): priority queue based on last access to frame (time when pin count goes to 0)
  - Based on theory that item which hasn't been accessed for a while is probably "no longer of interest"

Buffer Replacement Policy (Contd.)

- Policy can have big impact on # of I/O's; depends on the *access pattern*.
- *Sequential flooding*: Nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O.
  - Example scenario: join implementation with nested loops

Buffer Replacement Policies

- Lots of other replacement policies:
  - MRU
  - LFU (Least Frequently Used)
  - Random
  - FIFO (First In First Out)
  - Clock (Round Robin)
- Different benefits for different workloads
  - Also, some require keeping less state than others

- Disk Space Manager
- Buffer Manager
- Higher levels can simply request to read/ write a page without worrying whether it's in memory
  - Of course, they also need to notify the Buffer Manager when they no longer need the page

From Pages to Files

- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- As a first approximation, one DB relation = one file

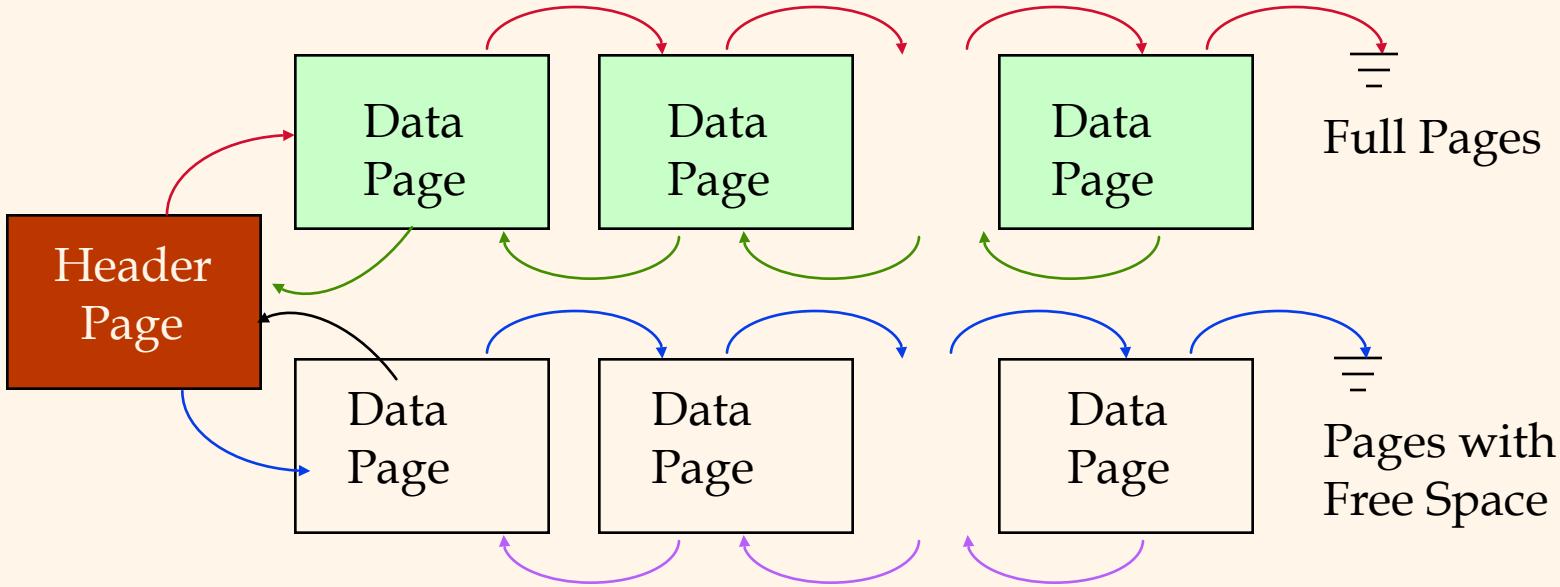
From Pages to Files

- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

Unordered (Heap) Files

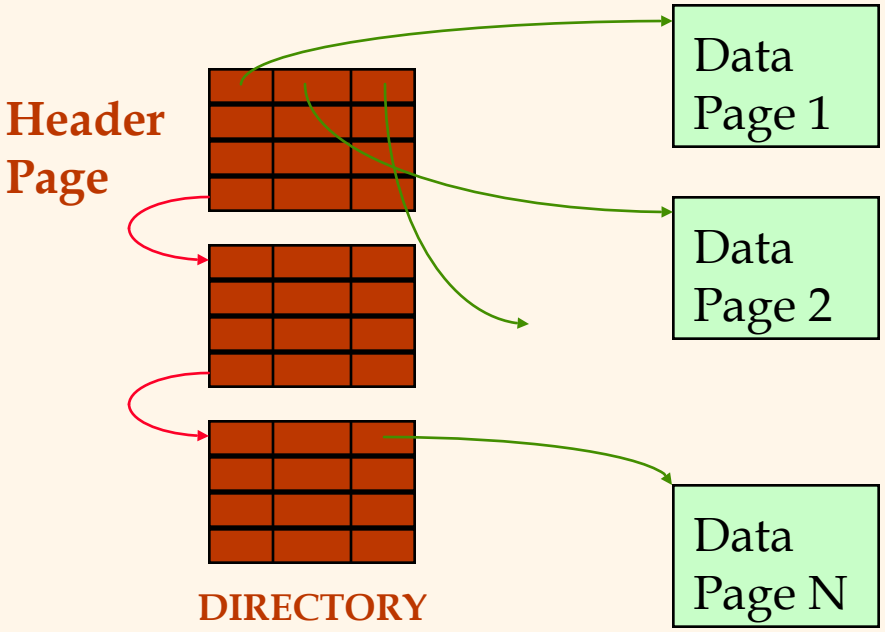
- Simplest file structure contains records in no particular order.
- As file grows and shrinks, pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this.

Heap File Implemented as a List



- The header page id and Heap file name must be stored someplace, e.g. in a **system catalog**.
- Each page contains 2 pointers plus data.

Heap File Using a Page Directory



- The entry for a page can include the number of free bytes on the page.
  - Easier to find a "good" page for insertion
- The directory is a collection of pages; linked list implementation is just one alternative.

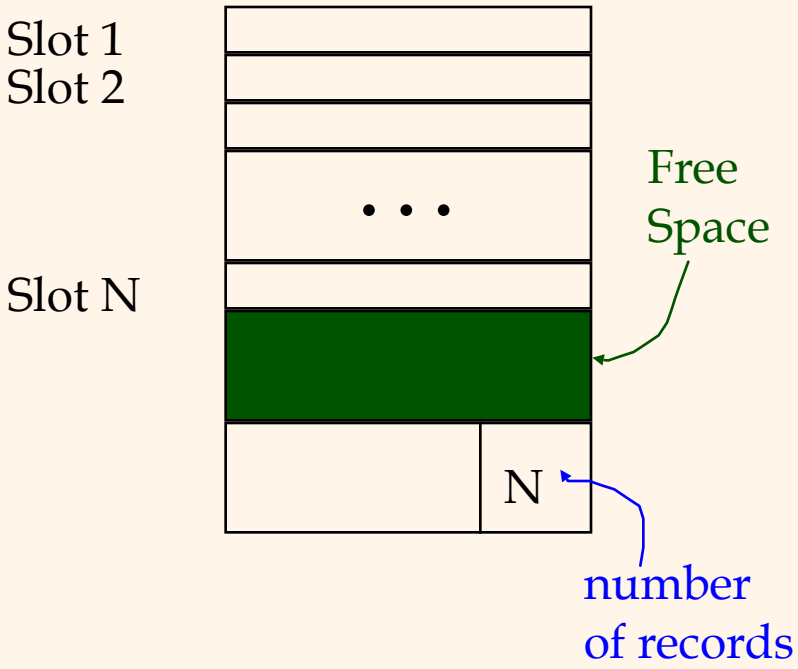
Next up: page formats

- How do we organize records on a page?
- Depends if the records are fixed or variable length
- Either way, page is considered as a set of **slots**
  - Each slot will contain a record
  - Each record can be identified by the pair <pageid, slotid>
    - We can just use that pair as the unique recordid (rid)

Fixed Length Records

- Division into slots straightforward
- Main issues to decide:
  - How to place records on the page
  - How to handle deletion

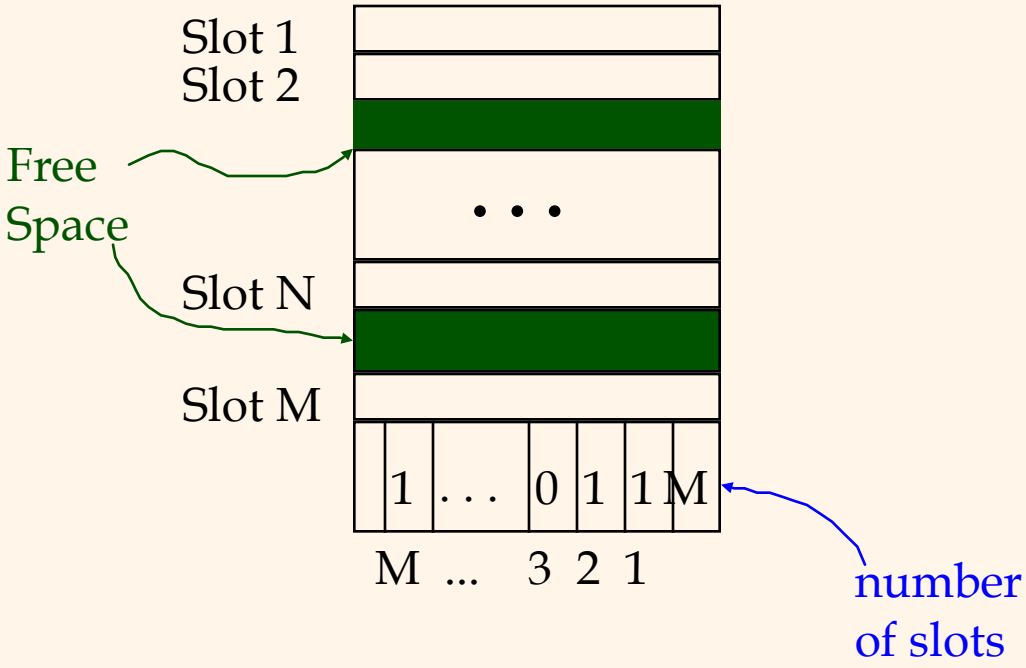
Option 1: packed organization



Option 1: "packed"

- All the nonempty records at the start of the page, free space at end
- Retrieving *record in i-th slot* is very easy, just an offset computation into the page
- Retrieving *all* records on a page also easy
- But must move around records upon deletion to perform compaction
  - May change rids if those include slot number
  - Not good if there are external references to these rids

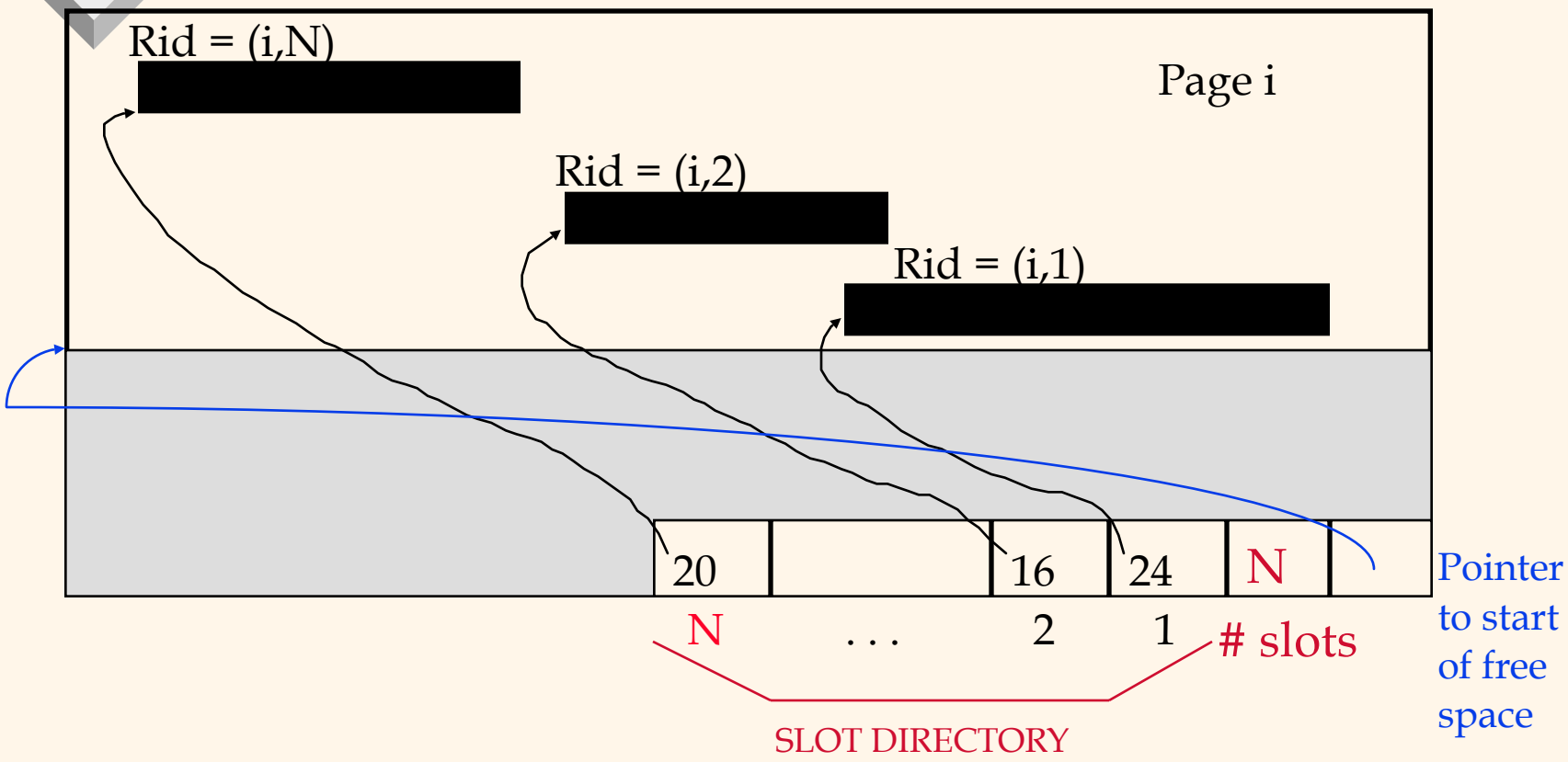
Option 2: "unpacked" with a bitmap



Variable length records

- Can't divide page into fixed slots up front
- Most flexible solution is to maintain a *slot directory*
  - Each slot is associated with an <offset, length> pair to indicate where the record can be found
  - Or, just store the offset in the slot and the length with the record (e.g. in the first few bytes)

Page Formats: Variable Length Records



- *Can move records on page without changing rid (only the offset stored in the slot changes)*

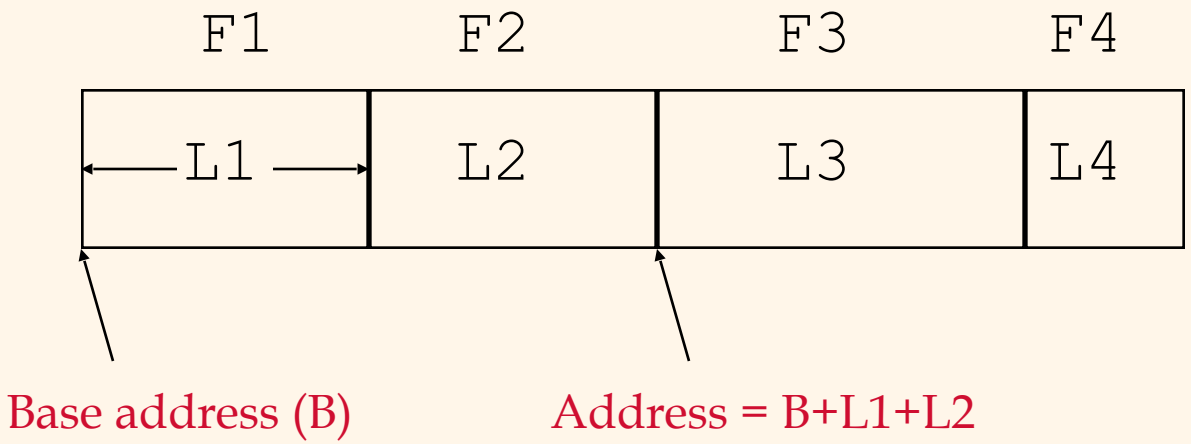
Managing free space

- Keep a pointer to start of free space area
- Periodically compact (move around records) to regain space
  - Either upon delete
  - Or upon insert if not enough space

Finally: record formats

- How to store fields within a single record?
- Again two options, based on whether records have fixed or variable length

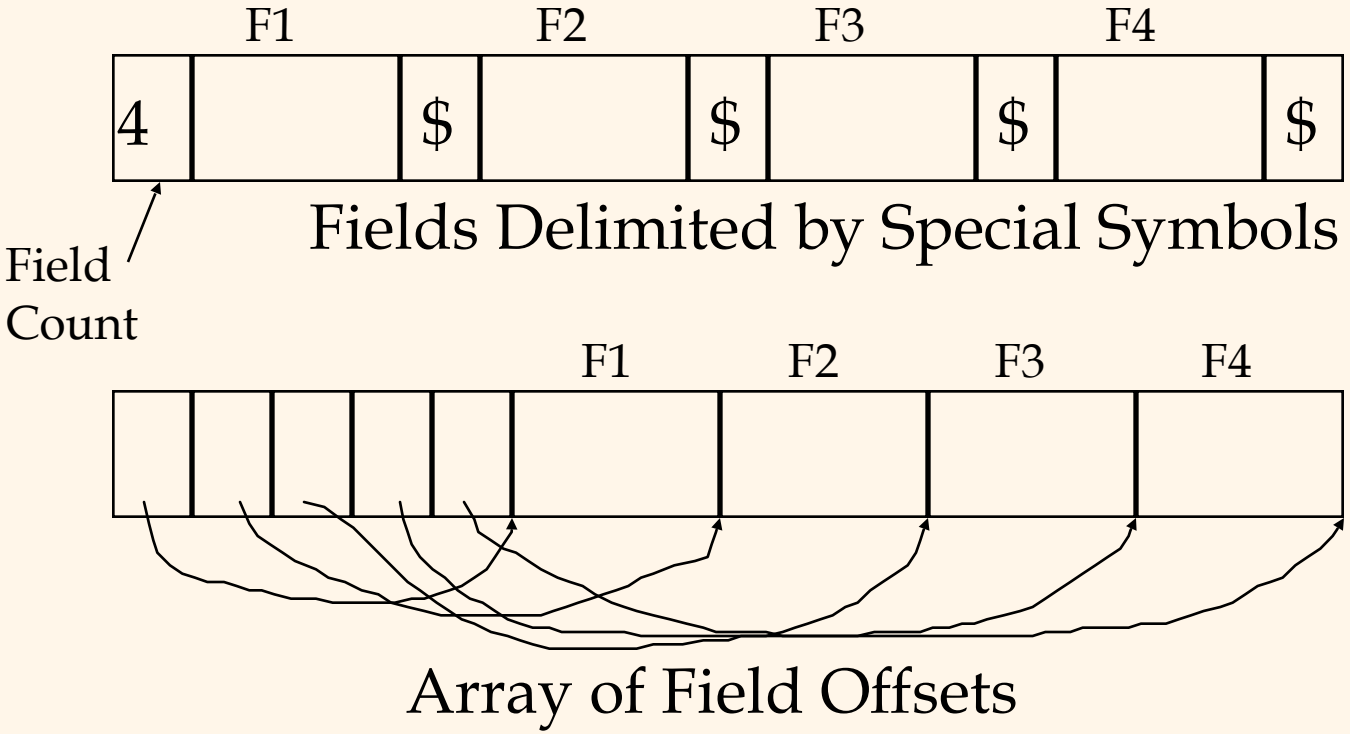
Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in ***system catalog***.

Record Formats: Variable Length

- Two alternative formats (# fields is fixed):



- Second offers direct access to i'th field, efficient storage of *nulls*; small directory overhead.

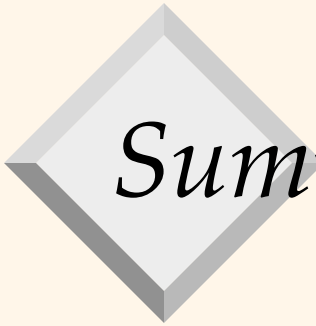
System Catalogs – Typical Info

- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

- *Catalogs are themselves stored as relations*!

MySQL system catalog

- INFORMATION_SCHEMA db
- Try the following (after "USE INFORMATION_SCHEMA;")
  - SHOW TABLES;
  - DESCRIBE Tables;
  - SELECT table_name FROM Tables;
  - SELECT * FROM Tables WHERE table_name='Boats' \G
  - SELECT * FROM Columns WHERE table_name='Boats' \G



Summary

- Storage management in a DBMS involves
- A Disk Space Manager
- A Buffer Manager
- A layer that provides the abstraction of a file of records
  - May be implemented on top of pages many ways, some of which we have discussed